Advanced C#

Joe Nalewabau
Program Manager
Visual C++
Microsoft Corporation

5-422


Microsoft®
PDC 2000
Professional Developers Conference
Microsoft®
.net™
the defining point

Advanced C# Programming

- Extending the type system
- Creating and using attributes
- API integration
  - DLL import
  - COM support
- C# and pointers

Extending The Type System

- Most users think of two types of objects
  - “Real” objects – Customer, Order
  - Primitive types – int, float, bool
- Different expectations for each
  - Real objects more expensive to create
  - Primitives always have a value
  - Primitives have operator support (+/- etc)

Traditional Solutions

1. Make everything an object
   - Performance implications
   - Behavior and expectation mismatch
2. Support both but restrict users to only “real” objects
   - Predefined primitive types with some inbuilt operator support
   - Users are locked out from creating primitive types

Rational Number

½, ¾, 1½

```
Rational r1 = new Rational(1,2);
Rational r2 = new Rational(2,1);

Rational r3 = r1.AddRational(r2);

double d =
Rational.ConvertToDouble(r3);
```

```
Rational r1 = new Rational(1,2);
Rational r2 = 2;

Rational r3 = r1 + r2;

double d = (double) r3;
```

Rational Number – Class

```
public class Rational
{
    public Rational(int n, int d)
{ … }
}
```

- Heap allocated
- Can be null
- “=” assigns reference not value
- Arrays allocates references not values
  - Rational[] array = new Rational[100];

Structs Provide The Answer

- Behavior differences versus Classes
  - Stored in-line, not heap allocated
  - Never null (always has default values)
  - Assignment copies data, not reference
- Implementation differences
  - Always inherit from object
  - Always has a default constructor

Rational Number – Struct

```
public struct Rational {
    public Rational(int n, int d)  { … }

    public int Numerator     { get{…} }
    public int Denominator { get{…} }

    public override string ToString()
{ … }
}
```

```
Rational r = new Rational(1,2);

string s = r.ToString();
```

Implicit Conversions

- No loss of data

```
public struct Rational {
    ...
    public static implicit operator
Rational(int i)
    {
        return new Rational(i,1);
    }
}
```

```
Rational r = 2;
```

Explicit Conversions

- Possible loss of precision and can throw exceptions

```
public struct Rational {
    ...
    public static explicit operator
double(Rational r)
    {
        return (double) r.Numerator /
r.Denominator;
    }
}
```

```
Rational r = new Rational(2,3);
 double d = (double) r;
```

Operator Overloading

- Static operators
- Must take its type as a parameter

```
public struct Rational {
    ...
    public static Rational operator+ (Rational
lhs,Rational rhs)
    {
        return new Rational(  ... );
    }
}
```

```
Rational r3 = r1 + r2;

r3 += 2;
```

Equality Operators

- .NET Framework equality support

```
public override bool Equals(object o)
```

- Usually overload == != to

```
public static bool operator== (Rational lhs,
Rational rhs)
public static bool operator!=  (Rational lhs,
Rational rhs)
```

```
if ( r1.Equals(r2) ) { … }   if ( !r1.Equals(r2)) { … }

if ( r1 == r2 ) { … }                    if ( r1 != r2 )  { … }
```

Structs And Interfaces

- Structs can implement interfaces to provide additional functionality
- Why? The same reasons classes can!
- Examples
  - System.IComparable
    - Search and sort support in collections
  - System.IFormattable
    - Placeholder formatting

System.IFormattable

- Types can support new formatting options through IFormattable

```
Rational r1 = new Rational(2,4);

Console.WriteLine(“Rational {0}",
      r1);
Console.WriteLine(“Rational
{0:reduced}", r1);
```

Implementing IFormattable

```
public struct Rational : IFormattable {
  public string Format(string formatStr,
IServiceObjectProvider isop) {
      s = this.ToString();
      if ( formatStr == “reduced" ) {  s = … }
      return s;
  }
}
```

```
Rational r1 = new Rational(2,4);

Console.WriteLine("No Format           = {0}", r1);
Console.WriteLine("Reduced Format = {0:reduced}",
r1);
```

```
No Format           = 2/4
Reduced Format = 1/2
```

Extending The Type System

Recap

- Best of both worlds!
  - Classes and Structs - No user lockout
- Natural semantics
  - Operator Overloading
  - User Conversions
- Interface support

Rational Number Demo

Advanced C# Programming

- Extending the type system
- Creating and using attributes
- API integration
  - DLL import
  - COM support
- C# and pointers

Attributes

- How do you associate information with types and members?
  - Documentation URL for a class
  - Transaction context for a method
  - Design time information
- Traditional solutions are disconnected
  - External files, e.g., .IDL, .DEF
  - Naming conventions between classes

Using Attributes

```
[HelpUrl("http://SomeUrl/APIDocs/SomeClass")]
class SomeClass
{
    [Obsolete("Use SomeNewMethod instead")]
    public void SomeOldMethod()
    { ... }

    public string Test( [SomeAttr()] string param1 )
    { ... }
}
```

Attribute Fundamentals

- Attributes are classes
  - Generic mechanism for users to add their own
- Attributes can be attached to types and members
  - Simple syntax to attach
- Attributes can be queried at runtime
  - Use reflection

Creating Attributes

- Attributes are simply classes
  - Derived from System.Attribute
  - Class functionality = attribute functionality

```
public class HelpUrlAttribute :
System.Attribute
{
   public HelpUrlAttribute(string url)
{ … }

   public string Url   { get {…} }
   public string Tag  { get {…} set {…}
```

Using Attributes

```
[HelpUrl("http://SomeUrl/MyClass")]
class MyClass  {}

[HelpUrl("http://SomeUrl/MyClass”,
Tag=“ctor”)]
class MyClass  {}

[HelpUrl("http://SomeUrl/MyClass"),
 HelpUrl("http://SomeUrl/MyClass”,
Tag=“ctor”)]
class MyClass  {}
```

Querying Attributes

- Use reflection to query attributes

```
Type type = typeof(MyClass);

foreach(object attr in
type.GetCustomAttributes() )
{
    if ( attr is HelpUrlAttribute )
    {
        HelpUrlAttribute ha = (HelpUrlAttribute)
attr;

        myBrowser.Navigate( ha.Url );
    }
}
```

Attribute Demo

Advanced C# Programming

- Extending the type system
- Creating and using attributes
- API integration
  - DLL import
  - COM support
- C# and pointers

Calling Into Existing DLLs

- .NET Framework contains attributes to enable calling into existing DLLs
- System.Runtime.InteropServices
  - DLL Name, Entry point, Parameter and Return value marshalling, etc.
- Use these to control calling into your existing DLLs
  - System functionality is built into

DLL Import Examples

```
[DllImport("gdi32.dll")]
public static extern
   int CreatePen(int style, int width, int
color);
```

```
[DllImport("gdi32.dll",CharSet=CharSet.
Auto)]
public static extern
   int GetObject(  int hObject,
                        int nSize,
                             [In, Out] ref LOGFONT
lf);
```

DLL Import Demo

Advanced C# Programming

- Extending the type system
- Creating and using attributes
- API integration
  - DLL import
  - COM support
- C# and pointers

COM Support

- .NET Framework provides great COM support
  - TLBIMP imports existing COM classes
  - TLBEXP exports .NET types
- Most users will have a seamless experience

COM Support

- Sometimes you need more control
  - Methods with complicated structures as arguments
  - Large TLB – only using a few classes
- System.Runtime.InteropServices
  - COM object identification

COM Support
Example

```
[Guid("56A868B1-0AD4-11CE-B03A-
0020AF0BA770")]
interface IMediaControl
{
  void Run();
  void Pause();
  void Stop();
  ...
  void RenderFile(string strFilename);
}
```

VideoPlayer Demo

Advanced C# Programming

- Extending the type system
- Creating and using attributes
- API integration
  - DLL import
  - COM support
- C# and pointers

C# And Pointers

- Developers sometime need total control
  - Performance extremes
  - Dealing with existing binary structures
  - Advanced COM Support, DLL Import
- C# “unsafe” = limited “inline C”
  - Pointer types, pointer arithmetic
  - unsafe casts
  - Declarative pinning (fixed statement)
- C# developers have headroom

C# And Pointers

```
struct COFFHeader {
    public ushort MachineType;
    public ushort
NumberOfSections;
    ...
    public ushort
Characteristics;
```

```
private COFFHeader fileHeader;

unsafe void ReadHeader(BinaryStream InFile)
{
   byte[] buffer =
InFile.ReadBytes(sizeof(COFFHeader));

   fixed (byte* headerPtr = buffer)
   {
       fileHeader = *((COFFHeader*)headerPtr);
    }
}
```

C# And Pointers

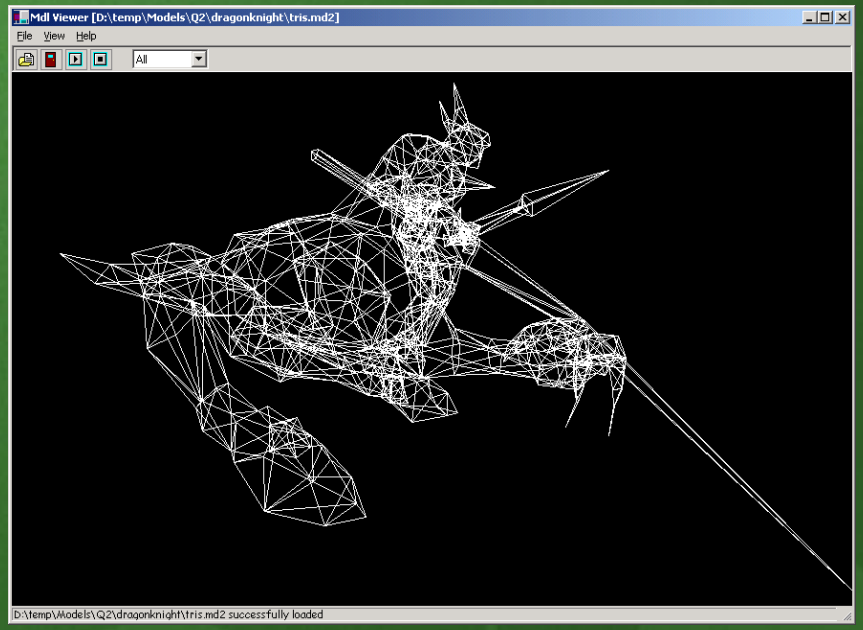
- Power comes at a price!
  - Unsafe means unverifiable code
  - Stricter security requirements
    - Before the code can run
    - Downloading code

Bringing It All Together

The “Intern” Project

- Build an application that requires:
  - Operator overloading
  - Attributes
  - COM Support
  - Unsafe
- Needs to be:
  - Visually appealing
  - Makes me look good ☺

DirectX® Demo



- Win Form Application
- DirectX® Surface
  - COM Support
- Win Form Events
  - C# calculations
  - DirectX® APIs to draw

Advanced C# Programming

- Extending the type system
- Creating and using attributes
- API integration
  - DLL import
  - COM support
- C# and pointers

Related Sessions And References

- Sessions:
  - Introduction to C#
  - Leveraging existing code in the .NET Framework
  - Writing self-describing applications on the .NET Framework
  - Debugging in Visual Studio® .NET
  - .NET Framework performance considerations
  - How does HelloWorld really work on the .NET Framework?
- References:
  - C# Language Reference
  - Hands On Lab session

Where do you want to go today?

Microsoft